Git

# 简介

Git 主张的分布式代码库与文件快照的设计思想，相对于传统 CVS、SVN 等集中式、文件差异式版本控制工具是一种挑战与颠覆。Git 带来了离线提交、轻量级分支等诸多便利。不过，也有人质疑 Git 的复杂性，并由此拔高了学习成本，某种程序上影响了开发者使用或者迁移 Git 的项目进度

Git 是一个开源的分布式版本控制软件。在英式英语中，Git 指一个愚笨或者不开心的人，恐怕与 Git 发明人——Linux 教父 Linus Torvalds 当时的自嘲心理不无关系吧。2002 年之前，Linux 内核维护工作的绝大部分时间都浪费在提交补丁与保存归档等繁琐事务上。启用版本控制工具 BitKeeper 管理 Linux 内核成了当务之急。不过，BitKeeper 毕竟是一款商业软件，在经历了 3 年免费使用之后，Linux 社区不得不寻求它的替代品，以便继续托管 Linux 内核源代码。2005 年，迫于无奈，Linus Torvalds 自行开发了一套开源版本控制工具，并命名为 Git。

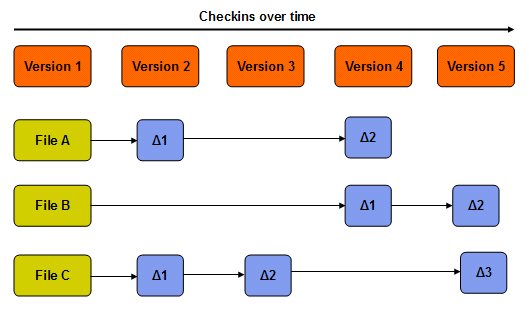
自诞生以来，Git 就以其开源、简单、快捷、分布式、高效等特点，应付了类似 Linux 内核源代码等各种复杂的项目开发需求。如今，Git 已经非常成熟，被广泛接受与使用，越来越多的项目都迁移到 Git 仓库中进行管理。以 Eclipse 社区为例。据称，目前 80% 的 Eclipse 基金会项目已经完全使用 Git 管理，CVS 访问权限已经切换成只读状态。并且，在 Eclipse 基金会官网中，针对项目管理的介绍中已将"CVS"三个字符划掉，而且很萌地写道，"Ding dong, the witch is dead."，意思是"叮咚，那个老巫婆已经挂了"。

# 优点

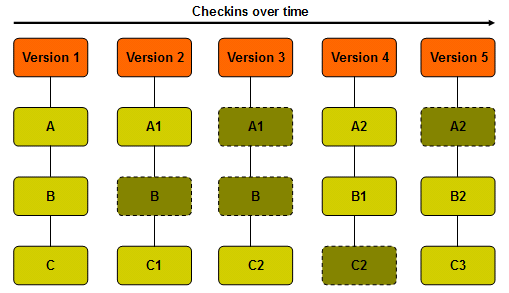
实际上，相对于 CVS、SVN 等主流版本控制软件，Git 的学习成本甚至会更高。比如，对于 Subversion 用户而言，如果能理解什么是文件、工作目录、资源库、版本、分支和标签等概念，差不多就够用了。而对于 Git 用户，需要理解更多更复杂的概念，包括文件、快照、工作树、索引、本地资源库、远程资源库、远程、提交、分支和 Stash 等。那么，为什么软件开发者对 Git 还是趋之若鹜呢？相比于 CVS 与 SVN，Git 的优势到底体现在哪里？

## A:

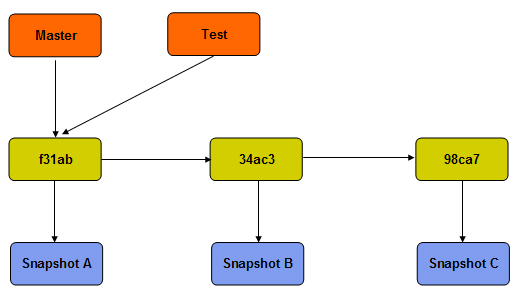
Git 底层自行维护的存储文件系统是一大亮点。CVS、SVN 底层采用的为增量式文件系统，如图 1 所示。增量式文件系统的特点是：当文件变动发生提交时，该文件系统存储的是文件的差异信息。



同样是文件变更提交，Git 底层文件系统存储的则为文件快照，即整个文件内容，并保存指向快照的索引，如图 2 所示。考虑到性能因素，如果文件内容没有发生任何变化，该文件系统则不会重复保存文件，只是简单地保存文件的链接。



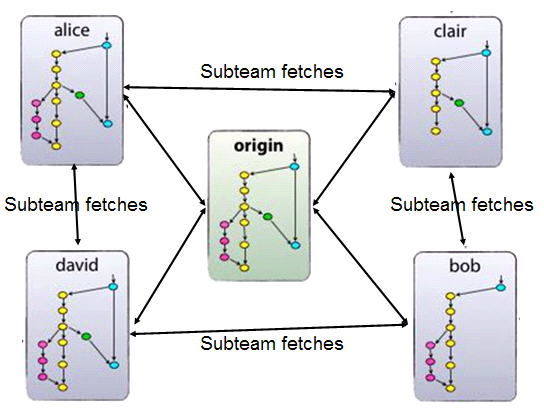
Git 之所以选择这样的底层存储数据结构，主要是为了提高 Git 分支的使用效率。实际上，Git 分支本质上是一个指向索引对象的可变指针，而每一个索引对象又指向文件快照

这样一来，创建分支可以瞬间完成，几乎不需要花费太多代价。换句话说，Git 分支是廉价的、轻量级的。我们看看各种 CVS、SVN 项目，分支通常意味着源代码的完整拷贝，其代价是昂贵的、重量级的。而对于大型项目来说，创建各种分支又是十分必要的，这与 Git 鼓励频繁创建与合并分支的理念相吻合。

## B:

Git 版本控制系统的设计思想是"去中心化"。传统的 CVS 、SVN 等工具采用的是 C/S 架构，只有一个中心代码仓库，位于服务器端。而一旦由于服务器系统宕机、网络不通等各种原因造成中心仓库不可用，整个 CVS 、SVN 系统的代码检入与检出就瘫痪了。即便考虑到高可用性，通过迁移另一个中心仓库继续代码提交操作，相应的运营维护成本也会随之上升。

为了摆脱对中心仓库的依赖，Git 的初始设计目标之一就是分布式控制管理。我们给出一个样例，如图 4 所示。假如我们成立一个项目组，开发者主要由 Alice、Bob、Clair、David 四名成员组成。其中，除了中心仓库 origin（Git 默认远程仓库名称）之外，每一名成员各自负责一个本地仓库。从分布式的观点来看，David 可看成是 Alice 的远程仓库，反过来也是一样。Git 分布式的设计理念有助于减少对中心仓库的依赖，从而有效降低中心仓库的负载，改善代码提交的灵活性。

****

Git 分布式设计思想所带来的另外一大好处是支持离线工作。离线工作的好处不言而喻，对于 CVS、SVN 这种严重依赖网络的 C/S 工具而言，没有了网络或者 VPN ，就意味着失去了左膀右臂，代码检入与检出操作就无法正常进行。而一旦使用 Git ，即便在没有 WIFI 的飞机或者火车上，照样可以频繁地提交代码，只不过先提交到本地仓库，等到了网络连通的时候，再上传到远程的镜像仓库。

# 如何安装 Git

总结起来，Git 安装方式通常分为两种：一种是选择 Git 源码编译安装；另一种使用针对特定平台的二进制安装包，又可以细分为 Linux、Mac、Windows 等，其安装说明如下。

## 在 Windows 上安装

# Git使用基础篇

Git是一个分布式的版本控制工具，本篇文章从介绍Git开始，重点在于介绍Git的基本命令和使用技巧，让你尝试使用Git的同时，体验到原来一个版 本控制工具可以对开发产生如此之多的影响，文章分为两部分，第一部分介绍Git的一些常用命令，其中穿插介绍Git的基本概念和原理，第二篇重点介绍 Git的使用技巧，最后会在Git Hub上创建一个开源项目开启你的Git实战之旅

## Git是什么

Git在Wikipedia上的定义：它是一个免费的、分布式的版本控制工具，或是一个强调了速度快的源代码管理工具。Git最初被Linus Torvalds开发出来用于管理Linux内核的开发。每一个Git的工作目录都是一个完全独立的代码库，并拥有完整的历史记录和版本追踪能力，不依赖 于网络和中心服务器。

Git的出现减轻了许多开发者和开源项目对于管理分支代码的压力，由于对分支的良好控制，更鼓励开发者对自己感兴趣的项目做出贡献。其实许多开源项目 包括Linux kernel, Samba, X.org Server, Ruby on Rails，都已经过渡到使用Git作为自己的版本控制工具。对于我们这些喜欢写代码的开发者嘛，有两点最大的好处，我们可以在任何地点(在上班的地铁 上)提交自己的代码和查看代码版本;我们可以开许许多多个分支来实践我们的想法，而合并这些分支的开销几乎可以忽略不计。

## Git 1+1

     现在进入本篇文章真正的主题，介绍一下Git的基本命令和操作，会从Git的版本库的初始化，基本操作和独有的常用命令三部分着手，让大家能够开始使用Git。

        Git通常有两种方式来进行初始化:

      git clone: 这是较为简单的一种初始化方式，当你已经有一个远程的Git版本库，只需要在本地克隆一份，例如'git clone git://github.com/someone/some\_project.git some\_project'命令就是将'git://github.com/someone/some\_project.git'这个URL地址的远程版 本库完全克隆到本地some\_project目录下面

git init和git remote：这种方式稍微复杂一些，当你本地创建了一个工作目录，你可以进入这个目录，使用'git init'命令进行初始化，Git以后就会对该目录下的文件进行版本控制，这时候如果你需要将它放到远程服务器上，可以在远程服务器上创建一个目录，并把 可访问的URL记录下来，此时你就可以利用'git remote add'命令来增加一个远程服务器端，例如'git remote add origin git://github.com/someone/another\_project.git'这条命令就会增加URL地址为'git: //github.com/someone/another\_project.git'，名称为origin的远程服务器，以后提交代码的时候只需要使用 origin别名即可

## Git的基本命令

现在我们有了本地和远程的版本库，让我们来试着用用Git的基本命令吧：

**git pull：**从其他的版本库(既可以是远程的也可以是本地的)将代码更新到本地，例如：'git pull origin master'就是将origin这个版本库的代码更新到本地的master主枝，该功能类似于SVN的update

**git add：**是将当前更改或者新增的文件加入到Git的索引中，加入到Git的索引中就表示记入了版本历史中，这也是提交之前所需要执行的一步，例如'git add app/model/user.rb'就会增加app/model/user.rb文件到Git的索引中

**git rm：**从当前的工作空间中和索引中删除文件，例如'git rm app/model/user.rb'

**git commit：**提交当前工作空间的修改内容，类似于SVN的commit命令，例如'git commit -m "story #3, add user model"'，提交的时候必须用-m来输入一条提交信息

**git push：**将本地commit的代码更新到远程版本库中，例如'git push origin'就会将本地的代码更新到名为orgin的远程版本库中

git log：查看历史日志

        git revert：还原一个版本的修改，必须提供一个具体的Git版本号，例如'git revert bbaf6fb5060b4875b18ff9ff637ce118256d6f20'，Git的版本号都是生成的一个哈希值、

        上面的命令几乎都是每个版本控制工具所公有的，下面就开始尝试一下Git独有的一些命令：

# GitHub入门

## Download and Install Git

At the heart of GitHub is an open source version control system (VCS) called Git\*. Created by the same team that created Linux, Git is responsible for everything GitHub related that happens locally on your computer.

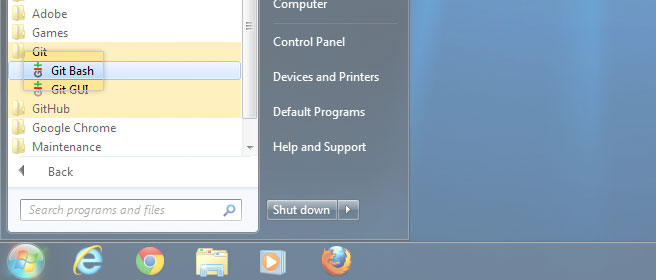
#### Download and install [the latest version of Git](http://git-scm.com/downloads).

<https://msysgit.googlecode.com/files/Git-1.9.0-preview20140217.exe>

## Set Up Git

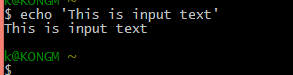
Now that you have Git installed, it's time to configure your settings. To do this you need to open Git Bash (not the Windows command line).

Code blocks like those on this page are part of a scripting language called Bash. To use Bash scripts, we need to use an application that was installed with Git called Git Bash. Git Bash can be found in the start menu under "Git".



#### Input

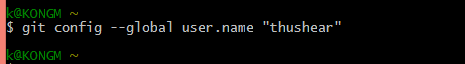
A line that begins with the dollar sign ($) indicates a line of Bash script you need to type. To enter it, type the text that follows the $, hitting the return key at the end of each line. You can hover your mouse over each line for an explanation of what the script is doing.



#### User Specific Input

#### Username

First you need to tell git your name, so that it can properly label the commits you make.



#### Email

Git saves your email address into the commits you make. We use the email address to associate your commits with your GitHub account.

Your email address for Git should be the same one associated with your GitHub account. If it is not, see [this guide](https://help.github.com/articles/how-do-i-change-my-primary-email-address) for help adding additional emails to your GitHub account. If you want to keep your email address hidden, [this guide](https://help.github.com/articles/keeping-your-email-address-private) may be useful to you.



#### Password caching

The last option we need to set will tell git that you don't want to type your username and password every time you talk to a remote server.

**Tip**: You need git **1.7.10** or newer to use the credential helper

To use this option, you need install a credential helper.

[GitHub for Windows](https://github-windows.s3.amazonaws.com/GitHubSetup.exe) includes this helper, and provides a git shell so you don't need to install and configure git manually.

**Tip**: The credential helper only works when you clone an HTTPS repository URL. If you use the SSH repository URL instead, SSH keys are used for authentication. [This guide](https://help.github.com/articles/generating-ssh-keys) offers help generating and using an SSH key pair.

### Celebrate

Congratulations, you now have Git and GitHub all set up! What do you want to do next?

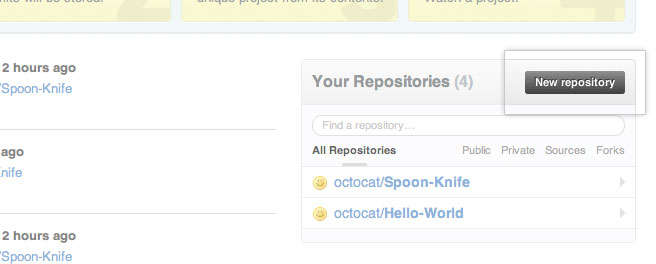
## **Create A Repo**

### Make a new repository on GitHub

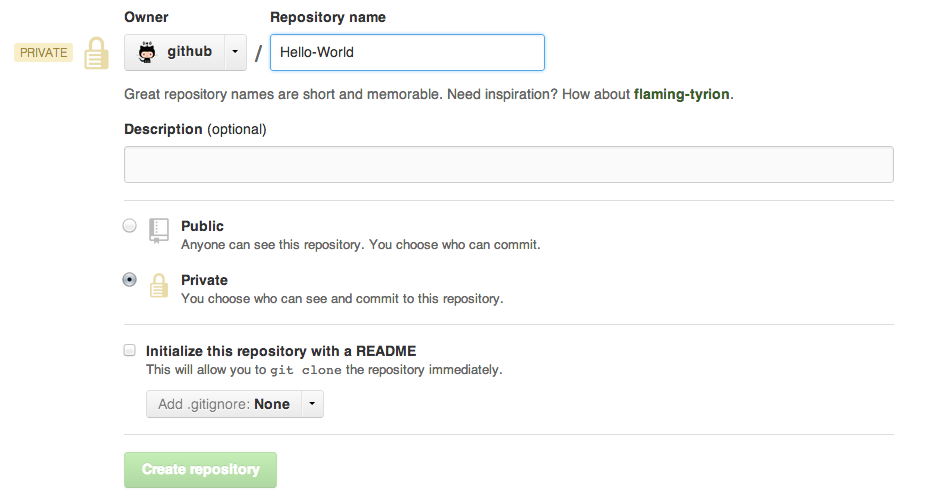
Every time you make a commit with Git, it is stored in a repository (a.k.a. "repo"). To put your project up on GitHub, you'll need to have a GitHub repository for it to live in.

Git stores all of your project files in a repository. If you are able to view hidden files on your system, you'll see a subdirectory called ".git" in the project directory where you run git init. This is where Git stores all of your commits, as well as everything else it needs. In addition to your local, you can also have remote repositories (like GitHub repos). Remote repositories are the same as your local repository, but stored on a different server or computer for easy collaboration, backup, and general awesomeness.

Click [New Repository](https://github.com/repositories/new).



Fill out the information on this page. When you're done, click "Create Repository."



Congratulations! You have successfully created your first repository!

### Create a README for your repository

While a README isn't a required part of a GitHub repository, it is a very good idea to have one. READMEs are a great place to describe your project or add some documentation such as how to install or use your project. You might want to include contact information - if your project becomes popular people will want to help you out.

If you include a file with the filename "README" in your repository, it will automatically be shown on your repository's front page. Pretty cool, huh? GitHub supports a number of different README formats. The one in this tutorial will result in a basic text file but other formats like .markdown or .textile can be used to render HTML content like links and headers. For more info about the supported markup formats, check out <https://github.com/github/markup>.

#### Step 1: Create the README file

In the prompt, type the following code:

mkdir ~/Hello-World

# Creates a directory for your project called "Hello-World" in your user directory

cd ~/Hello-World

# Changes the current working directory to your newly created directory

git init

# Sets up the necessary Git files

# Initialized empty Git repository in /Users/*you*/Hello-World/.git/

touch README

# Creates a file called "README" in your Hello-World directory

Open the new README file found in your Hello-World directory in a text editor and add the text "Hello World!" When you are finished, save and close the file.

#### Step 2: Commit your README

Now that you have your README set up, it's time to commit it. A commit is essentially a snapshot of all the files in your project at a particular point in time. In the prompt, type the following code:

Think of a *commit* as a snapshot of your project — code, files, everything — at a particular point in time. After your first commit git will only save the files that have changed, thus saving space.

**Be warned:** git will do it's best to compress your files, but large files and binaries can cause a repository to become bloated and unwieldy. Try to avoid committing things like compressed files (zips, rars, jars), compiled code (object files, libraries, executables), database backups, and media files (flv, psd, music, movies)

git add README

# Stages your README file, adding it to the list of files to be committed

git commit -m 'first commit'

# Commits your files, adding the message "first commit"

#### Step 3: Push your commit

So far, everything you've done has been in your local repository, meaning you still haven't done anything on GitHub yet. To connect your local repository to your GitHub account, you will need to set a remote for your repository and push your commits to it.

A remote is a repository stored on another computer, in this case on GitHub's server. It is standard practice (and also the default in some cases) to give the name origin to the remote that points to your main offsite repository (for example, your GitHub repository).

Git supports multiple remotes. This is commonly used when forking a repository.

git remote add origin https://github.com/*username*/Hello-World.git

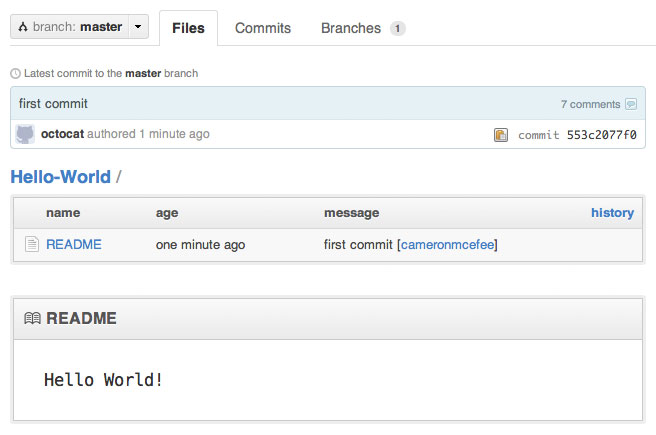
# Creates a remote named "origin" pointing at your GitHub repository

git push origin master

# Sends your commits in the "master" branch to GitHub

**Tip**: Notice that the path to your remote URL--Hello-World.git--matches the one that you created on GitHub. This is case sensitive, and important to keep the same.

Now if you look at your repository on GitHub, you will see your README has been added to it.



## **Fork A Repo**

### Contributing to a project

At some point you may find yourself wanting to contribute to someone else's project, or would like to use someone's project as the starting point for your own. This is known as "forking". For this tutorial, we'll be using the [Spoon-Knife](https://github.com/octocat/Spoon-Knife) project, hosted on GitHub.com.

#### Step 1: Fork the "Spoon-Knife" repository

To fork this project, click the "Fork" button in the GitHub.com repository.



#### Step 2: Clone your fork

You've successfully forked the Spoon-Knife repository, but so far it only exists on GitHub. To be able to work on the project, you will need to clone it to your local machine.

Run the following code:

git clone https://github.com/*username*/Spoon-Knife.git

#### Step 3: Configure remotes

When a repository is cloned, it has a default remote called origin that points to your fork on GitHub, not the original repository it was forked from. To keep track of the original repository, you need to add another remote named upstream:

A *remote* is a repository stored on another computer, in this case on GitHub's server. It is standard practice (and also the default when you clone a repository) to give the nameorigin to the remote that points to your main offsite repository (for example, your GitHub repository).

Git supports multiple remotes. This is commonly used when forking a repository.

cd Spoon-Knife

# Changes the active directory in the prompt to the newly cloned "Spoon-Knife" directory

git remote add upstream https://github.com/octocat/Spoon-Knife.git

# Assigns the original repository to a remote called "upstream"

git fetch upstream

# Pulls in changes not present in your local repository, without modifying your files

## More Things You Can Do

You've successfully forked a repository, but get a load of these other cool things you can do:

#### Push commits

Once you've made some commits to a forked repository and want to push it to your forked project, you do it the same way you would with a regular repository:

git push origin master

# Pushes commits to your remote repository stored on GitHub

#### Pull in upstream changes

If the original repository you forked your project from gets updated, you can add those updates to your fork by running the following code:

git fetch upstream

# Fetches any new changes from the original repository

git merge upstream/master

# Merges any changes fetched into your working files

There are two ways to get commits from a remote repository or branch: git fetch andgit pull. While they might seem similar at first, there are distinct differences you should consider.

##### Pull

git pull upstream master

# Pulls commits from 'upstream' and stores them in the local repository

When you use git pull, git tries to automatically do your work for you. It is context sensitive, so git will merge any pulled commits into the branch you are currently working in. One thing to keep in mind is that git pull automatically merges the commits without letting you review them first. If you don't closely manage your branches you may run into frequent conflicts.

##### Fetch & Merge

git fetch upstream

# Fetches any new commits from the original repository

git merge upstream/master

# Merges any fetched commits into your working files

When you git fetch, git retrieves any commits from the target remote that you do not have and stores them in your local repository. However, it does not merge them with your current branch. This is particularly useful if you need to keep your repository up to date but are working on something that might break if you update your files. To integrate the commits into your local branch, you use git merge. This combines the specified branches and prompts you if there are any conflicts.

#### Create branches

Branching allows you to build new features or test out ideas without putting your main project at risk. In git, branch is a sort of bookmark that references the last commit made in the branch. This makes branches very small and easy to work with.

Branches are pretty easy to work with and will save you a lot of headaches, especially when working with multiple people. To create a branch and begin working in it, run these commands:

git branch mybranch

# Creates a new branch called "mybranch"

git checkout mybranch

# Makes "mybranch" the active branch

Alternatively, you can use the shortcut:

git checkout -b *mybranch*

# Creates a new branch called "mybranch" and makes it the active branch

To switch between branches, use git checkout.

git checkout master

# Makes "master" the active branch

git checkout *mybranch*

# Makes "mybranch" the active branch

Once you're finished working on your branch and are ready to combine it back into themaster branch, use merge.

git checkout master

# Makes "master" the active branch

git merge *mybranch*

# Merges the commits from "mybranch" into "master"

git branch -d *mybranch*

# Deletes the "mybranch" branch

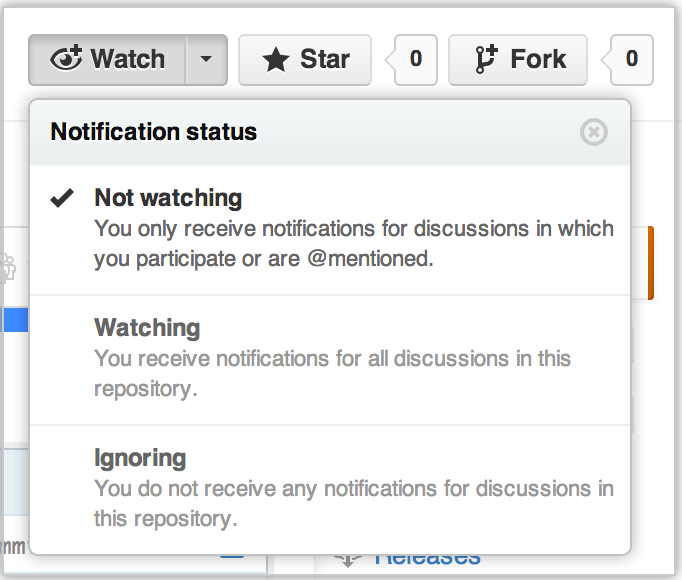
**Tip:** When you switch between branches, the files that you work on (the "working copy") are updated to reflect the changes in the new branch. If you have changes you have not committed, git will ensure you do not lose them. Git is also very careful during merges and pulls to ensure you don't lose any changes. **When in doubt, commit early and commit often.**

### Pull requests

If you are hoping to contribute back to the original fork, you can send the original author a [pull request](https://help.github.com/articles/using-pull-requests).

#### Unwatch the main repository

When you fork a particularly popular repository, you may find yourself with a lot of unwanted updates about it. To unsubscribe from updates to the main repository, click the "Unwatch" button on the **main repository** and select "Not Watching".



#### Delete your fork

At some point you may decide that you want to delete your fork. To delete a fork, just follow the same steps as you would to [delete a regular repository](https://help.github.com/articles/deleting-a-repository).

## **Be Social**

### Follow A Friend

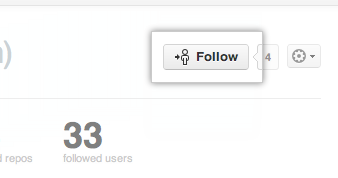
One of the great features on GitHub is the ability to see what other people are working on and who they are connecting with. When you follow someone, you'll get notifications on your dashboard about their GitHub activity.

#### Step 1: Pick a friend.

Why not follow one of these cool people from GitHub (including their pet tanuki!):

#### Step 2: Follow that friend (in a non-creepy way)

Once you are on one of their pages, click the "follow" button.



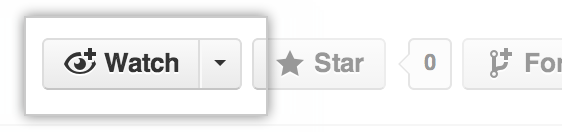
### Watch A Project

At some point you may want to stay up-to-date with a specific project. We've made this easy to do.

#### Watch a project

Our friend the Octocat has a project called [Hello World](https://github.com/octocat/Hello-World) that we'd like to watch.

Once you are on the project page, you will notice there is a "watch" button at the top of the page. Click on it.



Congratulations! You are now watching the Hello World project. If the Octocat updates it, you will see what happened in your dashboard.

## More Things You Can Do

You've done some of the most basic social interaction GitHub has to offer, but don't stop there! Check out these other social features:

#### Pull Requests



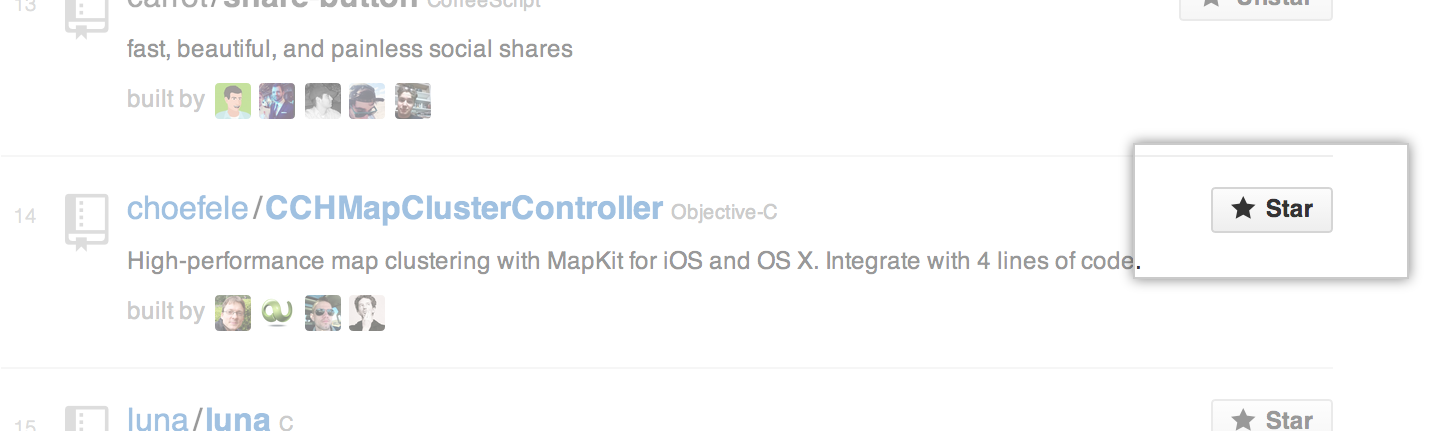
You may find yourself wanting to contribute to someone else's project, whether to add features or to fix bugs. After making changes, you can let the original author know about them by sending a [pull request](https://help.github.com/articles/using-pull-requests).

#### Issues



When you are collaborating on a project with someone, you sometimes come across problems that need to be fixed. To help you keep track of these problems, each GitHub repository has a section called Issues. For an example, check out the [issues](https://github.com/octocat/Spoon-Knife/issues) for the Spoon-Knife repository.

#### Explore



Discover interesting projects from across GitHub in the [Explore](https://github.com/explore) and [Trending](https://github.com/trending) sections. You can then star projects that you find interesting and want to come back to later—just visit your [Stars](https://github.com/stars) page to see all your starred projects.